

A Greedy Approach to Answer Reachability Queries on DAGs

Nicolas Boria Gianpiero Cabodi Paolo Camurati Marco Palena Paolo Pasini Stefano Quer

Dip. di Automatica ed Informatica, Politecnico di Torino, Torino, Italy

received yyyy-mm-dd, revised yyyy-mm-dd, accepted yyyy-mm-dd.

Several modern applications involve huge graphs and require fast answers to reachability queries. In more than two decades since first proposals, several approaches have been presented adopting on-line searches, hop labelling or transitive closure compression. Transitive closure compression techniques usually construct a graph reachability index, for example by decomposing the graph into disjoint chains. As memory consumption is proportional to the number of chains, the target of those algorithms is to decompose the graph into an optimal number c of chains. However, commonly used techniques fail to meet general expectations, are exceedingly complex, and their application on large graphs is impractical. The main contribution of this paper is a novel approach to construct such reachability indexes. The proposed method decomposes the graph into a sub-optimal number \hat{c} of chains by following a greedy strategy. We show that, given a vertex topological order, such a decomposition is obtained in $\mathcal{O}(\hat{c}m)$ time, and requires $\mathcal{O}(\hat{c}n)$ space, with \hat{c} bounded by $\lceil c \log(\frac{n}{c}) \rceil$. We provide experimental evidence suggesting that, on different categories of automatically generated benchmarks as well as on graphs arising from the field of logic synthesis and formal verification, the proposed method produces a number of chains very close to the optimum, while significantly reducing computation time.

Keywords: graph theory; DAG; reachability; transitive closure; data structure; greedy algorithm.

1 Introduction

The problem of answering reachability queries in an efficient way plays an important role in several modern applications. Domains in which this problem arises are as diverse as office systems, software management, geographical navigation, Internet routing and XML indexing. For example, in the domain of automated synthesis and verification of digital systems, both combinational and sequential circuits can be modelled, at some level of abstraction, as directed acyclic graphs, and it is often required to solve reachability queries on such graphs in order to discover variable dependencies or to compute variable scorings Baumgartner et al. (2013); Cabodi et al. (2016).

In order to answer reachability queries in constant time, a naïve approach would explicitly store the transitive closure of the graph, investing $\Theta(n^2)^{(i)}$ space and $\mathcal{O}(n^3)$ time. Such an approach would obviously

⁽ⁱ⁾ Given a graph $G = (V, E)$, we use n to denote the cardinality of the set of vertices $|V|$, and m to indicate the cardinality of the set of edges $|E|$.

be impractical for large graphs, which are often the case in several applications. For that reason, several more sophisticated approaches avoid storing the whole transitive closure of the graph. The majority of the existing reachability computation approaches belong to three main categories.

The first category Sanders and Schultes (2005); Trißl and Leser (2007); Yildirim et al. (2010) includes on-line searches. Instead of materializing the transitive closure, these methods use auxiliary labelling information for each vertex. This information is pre-computed and used for pruning the search space. For example, in GRAIL Yildirim et al. (2010) each vertex is assigned multiple interval labels where each interval is computed by a random depth-first traversal. The interval can help determine whether a vertex in the search space can be immediately pruned because it never reaches the destination vertex. The pre-computation of the auxiliary labelling information in these approaches is generally quite light, and the final index size is quite compact. Thus, these approaches can be applicable to very large graphs. However, the query performance is not appealing, and those methods can be easily one or two orders of magnitude slower than the ones belonging to the other two categories.

The second category Cohen et al. (2002); Ruoming and Guan (2013) includes reachability oracles, more commonly known as hop labelling. Each vertex v is labelled with two sets: $L_{out(v)}$, which contains hops (vertices) v can reach, and $L_{in(v)}$, which contains hops that can reach v . Given those two sets for each vertex v , and nothing else, it is possible to compute whether u reaches v by determining whether there is at least a common hop, $L_{out(u)} \cap L_{in(v)} = \emptyset$. Those methods lie in between the first category (on-line searches) and the third one (transitive closure materialization), as hop labelling can be considered as a factorization of the binary matrix of the transitive closure. Thus, it should be able to deliver more compact indices than the transitive closure and also offer fast query performance.

The third category Jagadish (1990); H. Wang and H. He and J. Yang and P. S. Yu and J. X. Yu (2006); Jin et al. (2008); Chen and Chen (2008, 2011); van Schaik and de Moor (2011) includes transitive closure compression approaches. This family of approaches aims to compress the transitive closure, i.e., to store for each vertex v a compact representation of $TC(v)$, e.g., all the vertices reachable from v . The reachability from vertex v to u is computed by checking vertex u against $TC(v)$. Representative approaches include chain compression, interval or tree compression, dual-labelling, path-tree, and bit-vector compression. Existing studies show how these approaches are the fastest in terms of query answering since checking against transitive closure $TC(v)$ is typically quite simple (linear scan or binary search suffices). However, the transitive closure materialization, despite compression, is still expensive, and the index size is often the reason these approaches are not scalable on large graphs. Moreover, pre-computation costs can be quite ineffective for certain applications. The amount of additional information associated to each node determines a trade-off between additional memory requirements and efficiency in answering reachability queries.

Jagadish Jagadish (1990) proposed an efficient data structure for answering reachability queries on DAGs, called *reachability index*. This data structure relies on a decomposition of the graph into \hat{c} disjoint chains and occupies $\Theta(\hat{c}n)$ space. In Jagadish (1990), the minimum number c of chains is proved to be equal to the width of the graph, e.g., the maximum number of nodes that are mutually unreachable. Jagadish proposed an algorithm to decompose the graph into the optimal number of chains that involves solving a min-flow problem. This algorithm runs in $\mathcal{O}(n^3)$ time. The author also proposes some heuristics to compute disjoint chain covers with a sub-optimal number of chains. Likewise, Felsner et al. (2003) suggest a path cover technique to solve the lowest common ancestor problem, and Kowaluk et al. (2008) develop a recognition algorithm for orders of small width and graphs of small Dilworth number based on similar ideas.

Chen et al. Chen and Chen (2008) improved on this result by proposing an algorithm able to decompose a graph into the optimal number of chains that requires only $\mathcal{O}(n^2 + cn\sqrt{c})$ time. Once the graph has been decomposed, labelling can be done in $\mathcal{O}(cm)$ time, as described in Chen and Chen (2008).

Chen et al. Chen and Chen (2011) propose a technique to answer reachability queries that relies partially on a tree decomposition of the graph and partially on a chain decomposition constructed as shown in Chen and Chen (2008).

Unfortunately, after more than two decades since first proposals, and a long list of worthy attempts, generally used techniques fail to meet general expectations or a exceedingly complex. In this paper, we follow the work by Jagadish (1990); Chen and Chen (2008, 2011); Teuhola (1996), and we design an algorithm adopting a simple and fast strategy which greedily decomposes the graph into disjoint chains. This is done by selecting the chain that includes the maximum number of nodes at each iteration. The technique achieves a good trade-off between space and time complexities, and this benefits are particularly useful when dealing with large graphs or in specific context, like in logic synthesis and formal verification, where only a limited amount of time can be dedicated to the pre-processing phase. The generated index can answer reachability queries as low as in constant time, depending on the way the labels associated to each node can be examined, thus satisfying time constraints of time-critical applications.

We demonstrate that our algorithm is able to construct a compact reachability index using $\mathcal{O}(\hat{c}n)$ time to decompose the graph into \hat{c} chains, given a topological ordering of the nodes. It also requires $\mathcal{O}(\hat{c}n)$ space, with \hat{c} bounded by $\lceil c \log(\frac{n}{c}) \rceil$, being c the minimum number of chains. Even though our approach leads to a non-minimum number of chains, our experiments suggest that its results are comparable to the optimum in terms of space, while significantly reducing the computation time. Furthermore, in addition to provide a different time/memory trade-off with respect to methods such as Jagadish (1990); Chen and Chen (2008, 2011), our technique is rather straightforward to implement.

1.1 Related Works and Comparison

The remainder of this paper is organized as follows. Section 2 illustrates the reachability index data structure proposed in Jagadish (1990). The algorithm we propose is described in Section 3. In Section 4 we prove time and space bounds of the proposed method. Section 5 presents an experimental evaluation on the algorithm over several categories of automatically generated benchmarks as well as on graphs arising from the field of logic synthesis and formal verification. Finally, in Section 6 we provide some summarizing remarks about the work.

2 DAG Reachability Index

2.1 Defining a Reachability Index

Let $D = (V, E)$ be a DAG⁽ⁱⁱ⁾, we add to V an artificial super-source s connected to all original sources, and an artificial super-sink t connected to all original sinks. Note that the mutual reachability of any pair of nodes in the graph is not affected by the introduction of such nodes. Nodes are arbitrarily numbered from 1 to n .

⁽ⁱⁱ⁾ For a cyclic graph $G = (V, E)$, i.e., a graph containing cycles, it is possible to find all strongly connected components (SCCs) in linear time Tarjan (1972). Those components can be collapsed into representative vertices, such that all nodes in an SCC are equivalent to their representative as far as reachability is concerned.

In Jagadish (1990), nodes of the DAG are subdivided into a set Π of disjoint reachability chains from s to t . A reachability chain π is defined as a sequence of nodes (v_1, v_2, \dots, v_k) such that, for every node $v_i \in \pi$, there exists a directed path in the graph from it to any of its subsequent nodes v_j , $i < j \leq k$. Each reachability chain $\pi \in \Pi$ is identified by a positive integer index starting from one. In order for such chains to be disjoint, each node of the graph must appear in exactly one of them, except for the artificial super-source s and super-sink t which appear in all of them. Each non-artificial node can thus be uniquely identified through a couple of indices (i, j) where i identifies the chain and j states the position of the node in such a chain.

The transitive closure of the DAG is materialized by labelling each of its nodes, except for s and t , with reachability information towards the computed chains. In particular, each node v is labelled with \hat{c} couples of indices (i, j) : \hat{c} is the number of chains, and each label (i, j) states that the highest node reachable from v in chain π_i is in position j . Note that for i equal to the chain of v , j corresponds to the direct successor of v in that chain. This way chains are implicitly stored as labels. We consider the labels associated to a node v as organized into an array λ_v of length \hat{c} . The i -th element of such an array corresponds to the position j of the highest node reachable from v in π_i . We consider the whole set of labels as organized into an array Λ indexed by positions in some pre-determined nodes ordering. The k -th element of Λ is λ_v , where v is the k -th node in such ordering.

The couple $I = (\Pi, \Lambda)$ forms a DAG reachability index. Storing such an index requires $\mathcal{O}(\hat{c}n)$ space. Since the number of considered chains is typically much lower than the number of nodes, the space required to store I is consequently much lower than the one required to explicitly store the transitive closure of the graph.

Example 1. In Figure 1, an example of a DAG reachability index is provided. Figure 1.a shows a simple DAG D . A possible reachability index for the graph, with 3 disjoint reachability chains (π_1, π_2, π_3) , is presented in Figure 1.b. Next to each node v , the couple of indices on top represents the node identifier (e.g., b is identified by $(2, 1)$, meaning that, in the second chain, b is the first node). The array of labels beneath it tracks the highest node reachable from v in every chain (e.g., the highest node reachable from b in the first chain is g , identified by the label $(1, 3)$).

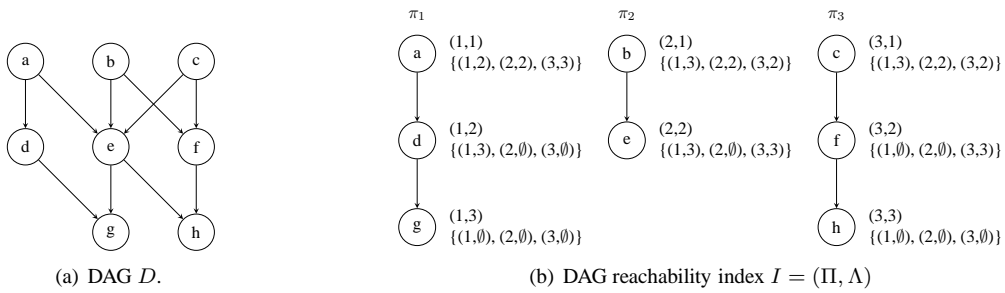


Figure 1: An example of DAG reachability index.

2.2 Building a Reachability Index

In the general case, nodes of a DAG $D = (V, E)$ can be subdivided into a variable number \hat{c} of disjoint reachability chains. An obvious upper bound for such number is n , i.e., the cardinality of the set of nodes $|V|$. In Jagadish (1990) it is demonstrated that the width of the graph c is a tight lower bound for \hat{c} . As mentioned earlier, different approaches are available to compute a set of disjoint chains Π of either optimal or sub-optimal size.

Once Π has been computed, nodes must be labelled. The procedure presented in Jagadish (1990) runs in $\mathcal{O}(m)$, and requires $\mathcal{O}(\hat{c}n)$ space. The labelling procedure is independent from the number of considered chains, and the way those chains were computed. In order to produce such a labelling, it is required to iterate over all outgoing edges of a node, updating an entry whenever a reachable vertex is found in a higher position with respect to the current state of a label, for a given path. Overall, this step requires to check all the edges of the original graph exactly once.

The main drawbacks of such algorithms are either the computational effort required to find the optimum or the size of the data structures used to store such a representation. Furthermore, depending on the chosen algorithm, query complexity may vary sensibly. Last, but not least, these kind of algorithms may require a significant effort in terms of implementation and optimization.

2.3 Querying DAG Using a Reachability Index

Given a DAG reachability index $I = (\Pi, \Lambda)$, it is possible to answer a reachability query between two nodes v, u in constant time. The related procedure is described by Algorithm 1.

For sake of simplicity, the predicates CHAIN and POSITION are used to retrieve some information about a node. CHAIN returns the index of the chain a vertex belongs to, and POSITION the position of the vertex within that chain. If v and u belong to the same chain, reachability from v to u can be decided by checking their relative position within that chain (line 2). Otherwise, if v and u belong to different chains, their reachability can be determined by comparing the position of the highest node reachable from v in the chain of u , to the position of u itself (line 4).

Input: $I = (\Pi, \Lambda)$, $v, u \in V$

Output: \top or \perp

```

1: if CHAIN( $v$ ) = CHAIN( $u$ ) then
2:    $answer \leftarrow$  POSITION( $v$ ) > POSITION( $u$ )
3: else
4:    $answer \leftarrow$  POSITION( $\lambda_v$ [CHAIN( $u$ )]) > POSITION( $u$ )
Return  $answer$ 

```

Algorithm 1: REACHABILITYQUERY

3 Algorithm Description

Given a DAG D , and a topological order of its nodes Γ , our algorithm produces a DAG reachability index in a simple, fast, and efficient way. The number of chains found is not the minimum one, but we will show

that, theoretically speaking, its upper bound is quite close to the exact value, and, practically speaking, its error is small in many real cases.

To generate the reachability index, we proceed in two steps, as shown by Algorithm 2. First, we compute a disjoint chain cover $\Pi = (\pi_1, \pi_2, \dots, \pi_k)$ of D (line 1). Then, we convert Π into a DAG reachability index by adding the appropriate reachability labels to each node $v \in V$ (line 2).

Input: $D = (V, E), s, t \in V, \Gamma$
Output: Π, Λ
 1: $\Pi \leftarrow \text{GREEDYCHAINCOVER}(D, s, t, \Gamma)$
 2: $\Lambda \leftarrow \text{NODESLABELLING}(D, t, \Pi, \Gamma)$
Return Π, Λ

Algorithm 2: DAGREACHABILITYINDEX

Before describing the algorithm in further details, we introduce the following definitions.

Definition 1. Let $D = (V, E)$ be a DAG, a chain π is either a single node $v \in V$ or a sequence of distinct nodes (v_1, v_2, \dots, v_k) , where v_{i+1} is reachable from v_i in D for each $1 \leq i \leq k-1$.

Definition 2. Let $D = (V, E)$ be a DAG and let $\Pi = (\pi_1, \pi_2, \dots, \pi_k)$ be a set of chains from s to t . A node $v \in V$ is said to be covered by Π iff $\exists \pi \in \Pi : v \in \pi$. Otherwise $v \in V$ is said to be non-covered by Π .

Definition 3. Let $D = (V, E)$ be a DAG and let $\Pi = (\pi_1, \pi_2, \dots, \pi_k)$ be a set of chains from s to t . Π is a chain cover of D iff every node $v \in V$ is covered by Π . Π is a disjoint chain cover of D iff it is a chain cover and each node belongs to exactly one $\pi \in \Pi$.

Definition 4. Let $D = (V, E)$ be a DAG and let $\Pi = (\pi_1, \pi_2, \dots, \pi_k)$ be a set of chains from s to t . The non-covered distance metric under Π is defined as the function $d : V \rightarrow \mathbb{N}$ that associates to each node $v \in V$ the maximum number of nodes non-covered by Π in any path from s to v .

The following two sub-sections describe procedures GREEDYCHAINCOVER and NODESLABELLING.

3.1 Greedy Chain Cover

Given Π and D , in order to compute the non-covered distance of each $v \in V$ under Π , we define a weight function $w : V \times V \rightarrow \{0, 1\}$ that assigns a Boolean value to each edge $(v, u) \in E$ as follows:

$$w(v, u) = \begin{cases} 0 & \text{if } u \text{ is covered by } \Pi \\ 1 & \text{otherwise} \end{cases}$$

Each edge is initialized with unitary weight. For each node $v \in V$, its non-covered distance under Π can be computed as the length of the longest path from s to v in the graph D weighted by w . Since D is a DAG, finding the longest path between two of its nodes can be done in $\mathcal{O}(n)$ time by taking into account the topological ordering of nodes. Computing such an ordering requires $\mathcal{O}(n + m)$ E. L. Lawler (1976).

Input: $D = (V, E)$, $s \in V$, $t \in V$, Γ

Output: Π

```

1:  $\Pi \leftarrow \emptyset$ 
2: while  $\exists v \in V$  non-covered do
3:    $\forall v \in V$  compute non-covered distance  $d(v)$  following  $\Gamma$ 
4:    $c \leftarrow t$ 
5:    $\pi \leftarrow \{t\}$ 
6:   while  $s \notin \pi$  do
7:      $p \leftarrow v \in V : (v, c) \in E \wedge d(v)$  is max
8:     if  $p$  is non-covered or  $p = s$  then
9:        $\text{UPDATE\_NODE\_INFO}(p, \pi)$ 
10:     $\pi \leftarrow \pi \cup p$ 
11:     $c \leftarrow p$ 
12:  $\Pi \leftarrow \Pi \cup \pi$ 
Return  $\Pi$ 

```

Algorithm 3: GREEDYCHAINCOVER

Algorithm 3 shows our procedure to compute a disjoint chain cover Π of the graph with a sub-optimal number of chains. Π is constructed incrementally, by adding a new chain to a partial cover at every iteration of the algorithm (line 12). Each chain is constructed backwards (lines 4-11), greedily selecting nodes based on their non-covered distance under the current Π . First the non-covered distance for each node is computed (line 3). Then, starting from t (lines 4-5), the algorithm iteratively extends the current chain π with a predecessor p of its last inserted node. This predecessor is the one whose non-covered distance $d(p)$ under Π is maximum (line 7). In order to produce a disjoint chain cover, upon selecting the next node p , we check whether it is already covered by Π or not (line 8). If p is not-covered, then we add it the current chain π (line 9) and the procedure continues. Upon introducing a node in a chain π , its local information is updated (line 10). Every node tracks the chain it belongs to, its position in π (with s starting at zero) and its successor, if any, within π . Otherwise such a node is not actively included in the chain π , but it is chosen nonetheless as the next step for chain construction (line 11). When s is reached, π is a chain from s to t with the highest number of nodes non-covered by the current Π . Note that the artificial nodes s and t are exceptions to the disjoint character of chains as they appear in each of them. The chain π is then added to the current partial cover Π . Next, the non-covered distance of every node is updated to match the current partial chain cover and the algorithm moves to the next iteration. Iterations proceed until every node has been covered by Π .

Example 2. Figure 2 provides an example in which our greedy approach is applied to a simple DAG.

At each iteration i , non-covered distances are updated and the chain π_i with longest non-covered dis-

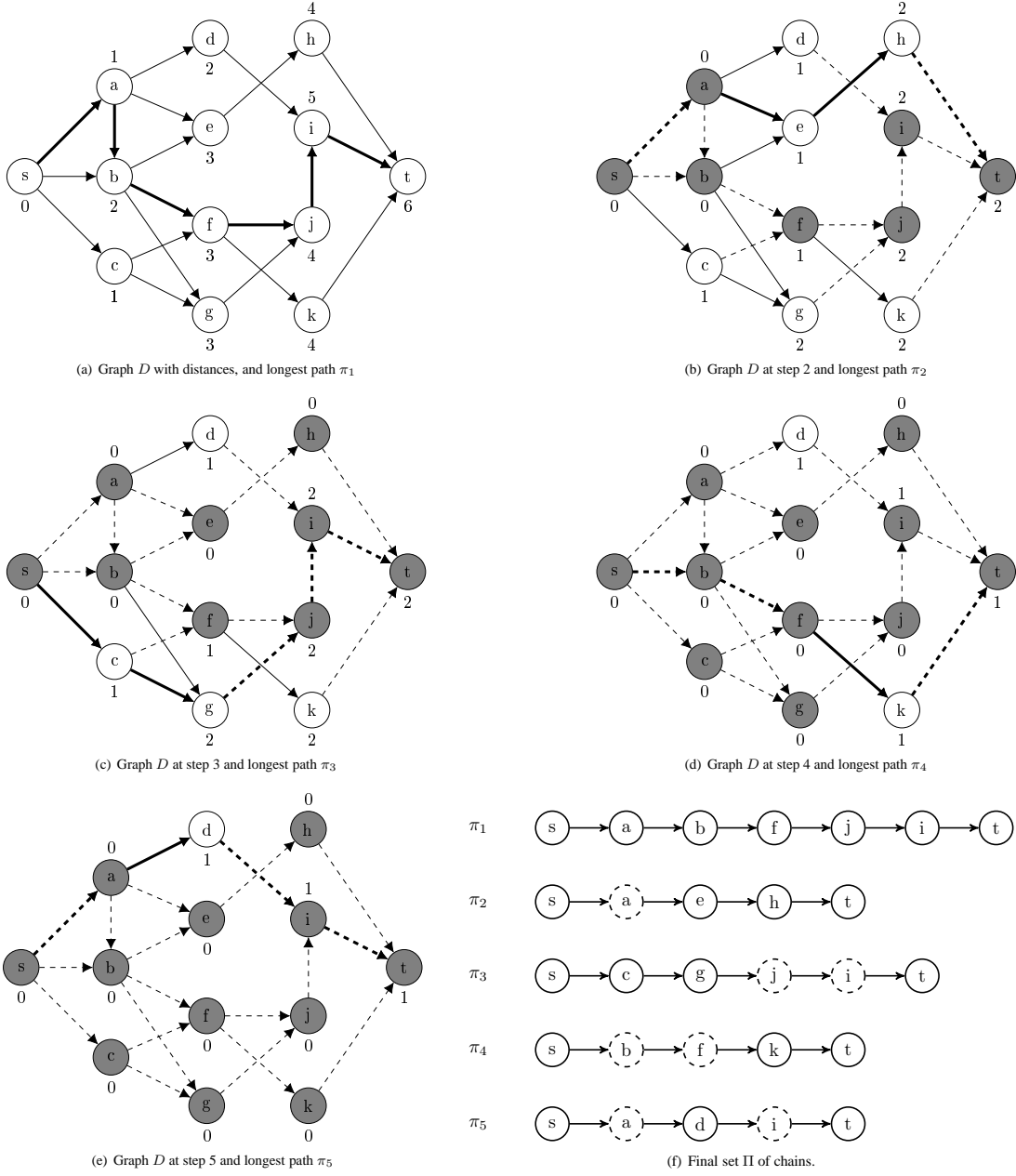


Figure 2: The algorithm running on a simple DAG

tance from s to t is computed. Edges with weight equal to zero are dashed. The chain constructed is represented with thicker edges. All newly covered nodes are shaded in the next sub-figure and all edges entering such nodes are given a weight equal to zero. Iterations continue until all nodes are covered by some chain.

Notice that the chain cover Π constructed by our algorithm consists of 5 paths $(\pi_1, \pi_2, \pi_3, \pi_4, \pi_5)$ reported in Figure 2.f, while an optimal one would require only 4 chains, e.g.,:

$$\begin{aligned}\pi_1 : & s \rightarrow b \rightarrow f \rightarrow k \rightarrow t \\ \pi_2 : & s \rightarrow c \rightarrow g \rightarrow j \rightarrow i \rightarrow t \\ \pi_3 : & s \rightarrow b \rightarrow e \rightarrow h \rightarrow t \\ \pi_4 : & s \rightarrow a \rightarrow d \rightarrow i \rightarrow t\end{aligned}$$

3.2 Nodes Labelling

Algorithm 4 shows the labelling procedure. In order to add cross-chain reachability information to each node we need to examine all of its outgoing edges. A single scan of the edges is able to address, in general, only direct reachability. In order to take into account also transitive reachability, we process nodes following backward the given topological order Γ (i.e., we adopt a reverse topological order). In this way we ensure that when processing a node, each of its descendants have been already labelled. We assume that for each node v , its label array λ_v is initialized with \hat{c} undefined labels. Undefined labels are denoted with the symbol \emptyset . We proceed by scanning each node in reverse topological order (line 1). Each non-artificial node is initialized with the set of non-undefined labels (if any) of its direct successor in its chain (line 5), while s and t are deliberately skipped (line 2-3). $\lambda_{\text{NEXT}(v)}$ is an acceptable set of labels for λ_v , because each node reachable from $\text{NEXT}(v)$ is also reachable from v .

Example 3. Taking into account the graph in Figure 2, the labelling procedure proceeds as follows. Nodes are processed in reverse topological order:

$$t \quad i \quad j \quad g \quad k \quad f \quad c \quad h \quad e \quad b \quad d \quad a \quad s$$

First of all, super-sink t is processed and skipped. Then, node i is processed. Since i is the non-artificial endpoint for chain π_1 no labelling information is inherited from its successor in π_1 . Since i can reach only t through a direct edge, no further labelling information is added to λ_i . The procedure continues on node j . λ_j is initialized with λ_i and the label for the direct successor i of j in chain π_1 is set. Since only the edge (j, i) exists in the graph and both i and j belong to the same chain, no further labels are updated. Node g is examined next. After the initialization of λ_g , the edge (g, j) is processed. Since the nodes g and j lie on different chains, and λ_g has no entry towards the first chain (where j lies in position 4), the entry $(1, \emptyset)$ is updated and becomes $(1, 4)$. Node k is examined next, and the algorithm behaves exactly as seen for node i . Node f is examined next, and the algorithms perform the same set of steps as seen for node j .

Procedure NODESLABELLING then proceeds in a similar way for the remaining nodes, finally produc-

Input: $D = (V, E)$, $s \in V$, $t \in V$, Π , Γ

Output: Λ

```

1: for each  $v \in V$  following  $\Gamma$  backwards do
2:   if  $v = t$  or  $v = s$  then
3:     continue
4:    $i \leftarrow \text{CHAIN}(v)$ 
5:    $u \leftarrow \text{NEXT}(v)$ 
6:    $\lambda_v \leftarrow \lambda_u$ 
7:    $\lambda_v[i] \leftarrow \text{POSITION}(u)$ 
8:   for each  $w \in V : (v, w) \in E$  do
9:     if  $w = t$  then
10:      continue
11:      $j \leftarrow \text{CHAIN}(w)$ 
12:     if  $i \neq j$  then
13:       if  $\lambda_v[j] = \emptyset$  or  $\text{POSITION}(w) < p$  then
14:          $\lambda_v[j] \leftarrow \text{POSITION}(w)$ 
15:    $\Lambda[v] \leftarrow \lambda_v$ 
Return  $\Lambda$ 

```

Algorithm 4: NODESLABELLING

ing the following labelling:

a :	(1, 1)	{(1, 2), (2, 1), (3, 2), (4, 1), (5, 1)}
b :	(1, 2)	{(1, 3), (2, 1), (3, 2), (4, 1), (5, \emptyset)}
c :	(3, 1)	{(1, 3), (2, \emptyset), (3, 2), (4, \emptyset), (5, \emptyset)}
d :	(5, 1)	{(1, 5), (2, \emptyset), (3, \emptyset), (4, \emptyset), (5, \emptyset)}
e :	(2, 1)	{(1, \emptyset), (2, 2), (3, \emptyset), (4, \emptyset), (5, \emptyset)}
f :	(1, 3)	{(1, 4), (2, \emptyset), (3, \emptyset), (4, 1), (5, \emptyset)}
g :	(3, 2)	{(1, 4), (2, \emptyset), (3, \emptyset), (4, \emptyset), (5, \emptyset)}
h :	(2, 2)	{(1, \emptyset), (2, \emptyset), (3, \emptyset), (4, \emptyset), (5, \emptyset)}
i :	(1, 5)	{(1, \emptyset), (2, \emptyset), (3, \emptyset), (4, \emptyset), (5, \emptyset)}
j :	(1, 4)	{(1, 5), (2, \emptyset), (3, \emptyset), (4, \emptyset), (5, \emptyset)}
k :	(4, 1)	{(1, \emptyset), (2, \emptyset), (3, \emptyset), (4, \emptyset), (5, \emptyset)}

Example 4. The graph in Figure 2 never leads the algorithms to perform the step at line 14 as a consequence of the second half of the *if* condition. To illustrate a label update in case of an improvement in terms of potential reachability, let us introduce another partial example, illustrated in Figure 3.

In such an example, we know that every node can reach its successor in its own chain, and also that *d* can reach *e* and therefore *a* can reach *e*, as well as that *c* can reach *b*. Assume that the edge (*a*, *c*) exists in

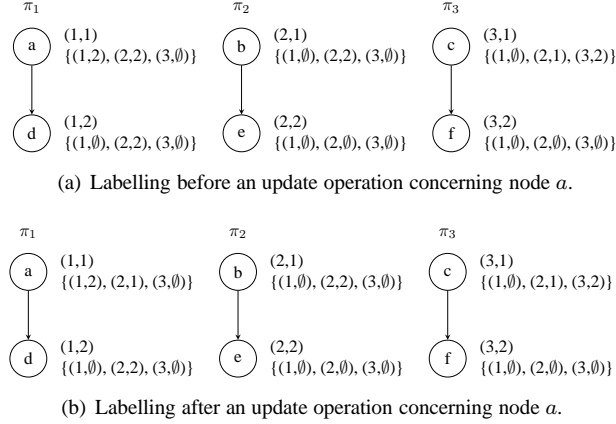


Figure 3: A second labelling example to illustrate solution improvements.

the original graph, the algorithm, whilst analyzing a , is going to recognize that the highest reachable node in chain π_2 from a can be updated. Originally a inherited reachability information from its successor d , but the label $(2, 2)$ for a can be improved given the existence of edge (a, c) , thus becoming $(2, 1)$.

4 Algorithm Analysis

As mentioned earlier, the algorithm GREEDYCHAINCOVER proposes a new trade-off between space and time complexities to compute and store the reachability index structure. More precisely, it generates a set of chains Π covering all nodes of the graph in a faster way than any previously known method, producing however a structure containing more chains, and thus larger indexes, than these methods.

In the following we analyze both aspects of this new trade-off. First, in Subsection 4.1, we provide a simple upper bound on the approximation ratio of the algorithm regarding the number of chains. Then, an analysis of the complexity of our algorithm is presented in Subsection 4.2. We also compare this complexity to those of the algorithms presented in Jagadish (1990) and Chen and Chen (2008).

4.1 Approximation Analysis

The algorithm GREEDYCHAINCOVER constructs a chain cover by greedily including in the solution the chain that contains the maximal number of non-covered vertices. In this sense the problem of finding a chain cover on a DAG D can be re-conducted to the classical and well studied SET COVER problem. In this interpretation, the elements to cover are the nodes of D , the sets covering the elements are all the chains from source to sink and the goal is indeed to cover all nodes using a minimum number of chains.

We provide an upper bound on the number of chains computed by the proposed algorithm by proving the following result.

Proposition 5. The output of GREEDYCHAINCOVER is a set of chains of cardinality at most $\lceil c \log(\frac{n}{c}) \rceil$ that covers every node of the DAG.

Proof: Let D be a DAG with a single source s and a single sink t that has n nodes. We wish to cover all nodes of D with as few chains from s to t as possible. Considering the nodes of D as a base set, and the

chains from s to t as sets of nodes, the problem at hand is equivalent to an instance I of the SET COVER problem. Moreover, we recall that at each of its iterations the algorithm GREEDYCHAINCOVER adds to the solution the chain that covers the maximal number of nodes not yet covered by the current solution. In this sense, GREEDYCHAINCOVER works in the exact same way as the classical greedy algorithm for SET COVER would on the equivalent instance I . This algorithm is known to be a $\log(\frac{n}{opt})$ -approximation algorithm, where n is the number of elements to cover and opt the cardinality of an optimal set cover (see for example Vazirani (2001) or Williamson and B.Shmoys and a slight refinement of the analysis found in Shuchi). In our case, the value of opt is known to be the width c of the graph, hence the solution computed by the greedy algorithm is $\log(\frac{n}{c})$ -approximate. In other words its cardinality is at most $\lceil c \log(\frac{n}{c}) \rceil$. \square

In section 5 we provide an experimental evaluation regarding the quality of this approximation, considering that $\lceil c \log(\frac{n}{c}) \rceil$ is merely an upper bound on the approximation ratio. Our experimental results suggest that the number of chains produced by GREEDYCHAINCOVER is actually very close to the optimum.

4.2 Complexity Analysis

In this subsection we show that GREEDYCHAINCOVER has lower time complexity than Chen and Chen (2008). As proved in the Subsection 4.1, the algorithm GREEDYCHAINCOVER computes a set of chains of cardinality at most $\lceil c \log(\frac{n}{c}) \rceil$. This also means that the algorithm will finish after at most $\lceil c \log(\frac{n}{c}) \rceil$ iterations. At each iteration, the algorithm updates the distance of each node in $\mathcal{O}(m)$ time and then computes the current longest chain also in $\mathcal{O}(m)$ time. Hence, the overall time complexity of GREEDYCHAINCOVER is $\mathcal{O}(mc \log(\frac{n}{c}))$.

	Topological Sort	Chains Construction	Labelling Time	Space Overhead
Jagadish (1990)	-	$\mathcal{O}(n^3)$	$\mathcal{O}(cm)$	$\mathcal{O}(cn)$
Chen and Chen (2008)	$\mathcal{O}(n + m)$	$\mathcal{O}(n^2 + cn\sqrt{c})$	$\mathcal{O}(cm)$	$\mathcal{O}(cn)$
GREEDYCHAINCOVER	$\mathcal{O}(n + m)$	$\mathcal{O}(\hat{c}m)$	$\mathcal{O}(\hat{c}m)$	$\mathcal{O}(\hat{c}n)$

Table 1: Comparison of space and time complexities.

In Table 1 we provide a comparison in terms of both time and space complexities among our method and those proposed in Jagadish (1990) and Chen and Chen (2008). Note that in our case, the labelling phase, although performed in a similar fashion as Chen and Chen (2008), involves a larger number \hat{c} of chains with respect to c . Namely $\mathcal{O}(cm)$, and answering a connectivity query using the labelled data structure only requires constant time. Of the three compared methods, only Jagadish (1990) does not require to previously compute a topological order of nodes. The space required to store the reachability index constructed by GREEDYCHAINCOVER is larger when compared to both Chen and Chen (2008) and Jagadish (1990), as it is proportional to the number of chains. Namely, each of the n nodes of the DAG is associated to a vector λ of labels that has as many elements as there are chains in the structure. Hence, instead of the $\mathcal{O}(cn)$ space required by a data structure with an optimal number c of chains, our structure needs $\mathcal{O}(\hat{c}n)$ space.

5 Experimental Results

In order to assess the applicability and performance of the proposed algorithm we devised a set of experiments taking into account different categories of graphs.

Experiments were run on an Intel Core i7-3770, with 8 CPUs running at 3.40 GHz, 16 GBytes of main memory DDR-III 1333, and hosting a Ubuntu 12.04 LTS 64 bits Linux distribution. The time limit was set to 7200 seconds (i.e., 2 hours), and the memory limit to 16 GB. We ran our algorithm over the selected set of benchmarks to compare the generated number \hat{c} of chains with the known optimum value c . The c value was computed deriving the length of the biggest anti-chain associated with our graph when interpreted as a *poset*, taking into account the theoretical aspect presented in Asratian et al. (1998). Such a procedure is quite straightforward as it simply requires to apply a maximum matching algorithms, such as Hopcroft–Karp, on an appropriately generated bipartite graph. Nevertheless, for very large graph the time required to perform this computation is non negligible, but we do not report evidence on that issue. Moreover, we report the time and memory require to run our algorithm. Section 5.1 also report some data to compare our methodology with the one reported in Chen and Chen (2008, 2011).

The experiments altogether tested five groups of data:

Dense graphs In this category fall graphs with a predefined number of nodes. Edges are added randomly until a certain density⁽ⁱⁱⁱ⁾ threshold (either 12.5% or 25% in our case) is reached.

Tree-based graphs This category includes graphs derived from a random tree, with a predefined number of initial nodes, each of which with a specific maximum degree. Such a tree is derived from a random Prüfer sequence. The final graph is the created adding a given number of additional edges.

Layered graphs In this category fall graphs generated as a sequence of layers, each characterized by a given number of nodes and for which each node has a specific maximum out-degree. An edge between two nodes is allowed only if such nodes reside in two contiguous layers.

General graphs To this category belong graphs generated randomly Erdős and Rényi (1959), considering one of the following criteria:

- Regularity: a d -regular random graph with a delimited number of nodes.
- Probability: a Erdős-Rényi graph with a predefined number of nodes, and an edge between each pair of nodes with a given probability.
- Cardinality: a random graph with a bounded number of nodes and edges.

Circuit-based graphs This category includes graphs derived from benchmarks of real-word devices usually adopted by the formal verification community. Given a design, we produce a benchmark by extracting the combinational portion of the sequential circuit. Benchmarks were selected in order to include a significant range of variability in terms of design characteristics.

Notice that the first four graph sets are the ones introduced by Chen and Chen (2011). For sake of generality, we generate those graphs with the publicly available tool `NetworkX` Edwards. Circuit-based benchmarks are obtained from the `HARDWARE MODEL CHECKING COMPETITION` suites Biere et al., and mostly come from industrial designs. We include this category to demonstrate the applicability of

⁽ⁱⁱⁱ⁾ We define the density of a graph as m/n^2 , and we usually specify it as a percentage value.

the proposed method on practical tests from the formal verification community, rather than just limiting ourselves to experiments on randomly generated benchmarks. The specific characteristics of each experimental instance, such as number of vertices, edges, density, etc., are reported in the following sections, alongside the results obtained.

5.1 Comparison

As a preliminary set of experiments, we compare our technique with our implementation of the exact one presented in Chen and Chen (2008). The original description reported on Chen and Chen (2008) does not completely depict how the algorithm behaves in some limit cases that caused the procedure to generate incorrect, that is, non-optimum, results. To address this issue, we took into account unreferenced material published by the same authors. With our final implementation, we were not able to obtain times comparable with those proposed in Chen and Chen (2008), and experiments on graphs larger than a few thousand nodes usually did not terminate within the slotted time limit, despite the fact that we run our experiments on a significantly more performing hardware platform. As a consequence, we restrict our comparison on a set of small dense graphs such as the ones reported by Chen and Chen (2008). Table 2 summarizes those results.

Table 2: Experimental results on small dense graphs.

n	m	Density	Depth	Width	Chains	ϵ	Time [s]	
		[%]				[%]	Greedy	Chen and Chen (2008)
1000	125000	25	342	10	13	30.00	0.47	6.95
1000	125000	25	339	9	11	22.22	0.50	6.69
2000	500000	25	668	10	13	30.00	0.87	33.37
2000	500000	25	683	9	12	33.33	0.70	31.22
3000	1250000	25	1043	10	13	30.00	1.23	107.30
3000	1250000	25	1056	11	12	9.09	1.32	94.77

Columns n and m report the number of vertices and edges in the graph. For each instance, we indicate its density, depth and width (corresponding to the minimum number of chains c). Not surprisingly, dense graphs present a rather low width as the high number of connections among vertices limits the number of mutually unreachable nodes. As expected, on graphs with low widths our procedure yields rather high relative error values. Column Chains report the \hat{c} value, and column ϵ the relative error $((\hat{c} - c)/c \cdot 100)$. The relative high error is due to the greedy behaviour of our approach when running on those graphs. The algorithm has little room to recover from sub-optimal choices made during its first steps given the low optimum number of chains. At the same time, though, such a low optimum renders the impact of the relative error rather inconsequential in practice as the impact on the memory usage is quite moderate. The

graphs with 3000 vertices have the same density of the ones used by Chen and Chen (2008, 2011). In Chen and Chen (2008) the authors present an average labelling time for these graphs of about 20 seconds. Our implementation of their algorithm is able to deal with those graphs in about 100 seconds, on average. As previously noticed, our hardware platform should be faster than the one used by the original authors, then the difference may be due to some undefined shortcuts of our implementation or some optimizations not disclosed by the authors. In any case, our method perform labelling in a little more than a second. Then it should be at least about one order of magnitude faster than the original algorithm presented in Chen and Chen (2008). Similar considerations hold for a direct comparison with Chen and Chen (2011), where the authors, using spanning trees instead of chains, again on a similar set of graphs, present labelling time in the order of 15 seconds.

5.2 Randomly generated graphs

In this subsection we present experimental results concerning randomly-generated graph instances. In order to test the proposed techniques on benchmarks with different topologies and characteristics, we take into account the four graph categories previously introduced. For each category, we provide a dedicated section, reporting the size and characteristics of each benchmark instance together with the results obtained with the proposed technique, i.e., the number of generated chains (\hat{c}) and the actual graph width (c). We also report execution time and memory for each run.

As far as memory usage is concerned, notice that adopting the implementation depicted in Section 2.3, each chain entry can be represented by an integer value. Then, the memory usage is the one necessary to store a number of integer values equal to $n\hat{c}$. Chen et al. Chen and Chen (2008, 2011) use 16-bit integer values to store chains. On the contrary, as we manipulate graphs larger than 65536 vertices, we use 32-bit integer values. Further possible optimizations, e.g., data compression schemes such as the one proposed by van Schaik et al. van Schaik and de Moor (2011), are not taken into account in the tables.

5.2.1 Dense graphs

Table 3 illustrates the results on dense graphs larger than the ones considered in Section 5.1. Dense graphs tend to be characterized by low values of width: Given the high connectivity among nodes, it is unlikely to have a large subset of mutually unreachable nodes. For this reason, the error can potentially grow quite large, since the proposed greedy approach terminates in a small number of steps, and subsequently has fewer chances to recover from sub-optimal choices. At the same time, such an error is relative to a small number of chains, thus its impact (in term of memory usage) is quite small in practice.

Table 3 reports graphs with increasing size, considering for each size two density values, i.e., about 12.5% and 25.0%. Results in terms of width, chains, errors and times are pretty regular, then we just report results for 10 different graphs. All running times remain in the order of a few seconds, and memory usage stays below 1 MBytes for all cases.

5.2.2 Tree-based graphs

In this subsection we present experimental results concerning tree-based instances. Such a set of graphs is characterized by a peculiar topology and structure. Since they are derived from a tree, these graphs tend to get wider as the depth (in terms of distance from the root) increases. This unusual structure is rather problematic for techniques such as Chen and Chen (2008), as it force the algorithms to introduce a significant number of additional nodes during the chain generation phase. On the contrary, our proposed

Table 3: Experimental results on dense graphs.

n	m	Density [%]	Depth	Width	Chains	ϵ [%]	Time [s]	Memory [MB]
2002	500008	12.48	684	11	13	18.18	0.49	0.10
2002	1000004	24.96	1146	6	7	16.67	0.87	0.05
4002	2000009	12.49	1385	10	12	20.00	1.85	0.18
4002	4000006	24.98	2278	7	7	0.00	2.44	0.11
6002	4500009	12.49	2069	11	13	18.18	3.04	0.31
6002	9000005	24.99	3446	7	7	0.00	4.16	0.16
8002	8000010	12.50	2726	11	13	18.18	4.48	0.42
8002	16000003	24.99	4573	7	7	0.00	6.99	0.21
10002	12500009	12.50	3441	11	14	27.27	7.07	0.53
10002	25000005	24.99	5838	7	7	0.00	11.52	0.27

technique generate chains which share many nodes in the narrower part of the graphs and then spread out following the branches of the tree.

Table 4 shows our results.

The meaning of the columns is the one described for Table 2. Errors are much smaller than for dense graphs, and they tend to decrease for larger graphs where our greedy approach seems to close the gap with the exact technique in terms of accuracy. Our memory usage is then almost identical to the one required by the exact technique. In both cases large values of memory usage are due to the high number of chains found for those graphs, since, as previously mentioned, memory usage is in the order of $\mathcal{O}(\hat{c}n)$, and it is the intrinsic limitation of transitive closure compression techniques. Luckily, the time required to generate the index is much smaller than exact techniques being in the order of tens of seconds for all benchmarks but the largest instances.

5.2.3 Layered graphs

Layered graphs are characterized by a regular structure, in which nodes are arranged in layers each presenting the same number of nodes. Nodes in each layer can only reach nodes in the directly subsequent layer. This kind of structure intrinsically favours matching-based algorithms for transitive closure compression. The number of nodes in each layer is a trivial lower bound to the width of the graph itself, and thus to the number of chains. Furthermore, chains in this kind of graphs, which are usually limited in terms of depth, tend to be short.

The experimental evaluation shows that the proposed greedy technique produces results that are sometimes quite far from the optimum. However, we can identify a decreasing trend of the relative error as the number of edges increases. Intuitively, the graph structure easily leads the greedy algorithm to several local minimum. This is due to the fact that a choice at a certain level is definitive and has a direct impact

Table 4: Experimental results on graphs derived from trees.

n	m	Density [%]	Depth	Width	Chains	ϵ [%]	Time [s]	Memory [MB]
5004	16825	0.067	767	670	753	11.02	1.52	14.37
5004	31732	0.127	937	319	393	18.83	1.08	7.50
10004	23635	0.024	1075	1801	1982	9.13	5.51	75.65
10004	28648	0.029	1131	1496	1697	11.84	4.77	64.76
10004	33607	0.034	1043	1255	1440	12.85	4.78	54.95
15004	30442	0.014	986	3264	3490	6.48	16.32	199.75
15004	35513	0.016	1235	2682	2973	9.79	14.18	170.16
15004	40536	0.018	1319	2412	2703	10.77	11.46	154.71
20004	37343	0.009	1045	5014	5273	4.91	25.29	402.38
20004	42344	0.011	1242	4182	4511	7.29	24.66	344.23
50004	78431	0.003	1566	15634	15970	2.10	300.61	3046.28
100004	151242	0.002	2339	19343	19615	1.40	911.39	7482.83
150004	403242	0.002	2986	23277	23435	0.68	1431.92	13409.97

on the set of reachable nodes on the layers above. As the number of edges increases, a sub-optimal choice at a certain level can still be improved given the higher number of alternative paths among nodes.

Table 5 illustrates the results on layered graphs. For time and memory figures it is possible to make similar considerations to the one presented in Section 5.2.2.

5.2.4 General graphs

As a last set of randomly-generated graph, we take into account completely random graphs created with two different generation models. In the first model, we pre-define both the number of nodes and edges, selecting a graph with such numbers of nodes and edges with equal probability among all possible graphs with those characteristics. Alternatively, in the second model, we specified just the number n of vertices, and then an edge was added to the graph, between any two given nodes, with a chosen probability.

Table 6 illustrates the results on general graphs. Density is usually quite small, being those graph relatively sparse. Exact (column Width, c) and over-estimated chains (column Chains, \hat{c}) present a very high variance. Similarly, it is possible to notice a wide range of relative errors. The higher values are associated with graphs with lower widths, whereas relative errors tends to decrease for larger graphs. From the experimental results, it is possible to appreciate the applicability of the proposed technique on graphs up to 100,000 nodes, without an inordinate computational effort.

Table 5: Experimental results on layered graphs.

n	m	Density [%]	Depth	Width	Chains	ϵ [%]	Time [s]	Memory [MB]
2502	12007	0.192	6	508	623	22.64	0.94	5.95
5002	23763	0.095	6	1009	1260	24.88	1.64	24.04
5002	25517	0.102	11	507	651	28.40	1.38	12.42
5002	47428	0.190	11	502	582	15.94	1.35	11.11
5002	112185	0.448	11	501	535	6.79	1.99	10.21
7502	38909	0.069	16	506	650	28.46	1.58	18.60
7502	73523	0.131	16	503	584	16.10	2.26	16.71
7502	90678	0.161	16	502	565	12.55	2.06	16.17
10002	29882	0.030	21	549	785	42.99	2.32	29.95
10002	51254	0.051	11	1014	1300	28.21	3.91	49.60
10002	53093	0.053	21	505	645	27.72	2.64	24.61
10002	95505	0.095	11	1003	1157	15.35	5.03	44.14
10002	226435	0.226	11	1001	1075	7.39	7.53	41.02
15002	79005	0.035	16	1009	1294	28.25	7.61	74.05
15002	146627	0.065	16	1005	1166	16.02	8.75	66.73
15002	182637	0.081	16	1003	1130	12.66	9.24	64.67
20002	59806	0.015	21	1083	1549	43.03	8.94	118.19
20002	105982	0.026	21	1014	1303	28.50	8.93	99.42

5.3 Circuit-based graphs

In this subsection we present experimental results concerning circuit-based instances. These kind of circuit are derived from real-life benchmarks usually adopted in the field of hardware formal verification. To apply Model Checking on these benchmarks it is usually required to run a large variety of pre-processing steps with various techniques. Many of those techniques explicitly require mutual vertex reachability, eventually restricted to specific sub-set of vertices Baumgartner et al. (2013); Cabodi et al. (2016), such as the ones representing primary inputs, primary outputs, or present or next state variables. As a consequence, this section provides some sort of case study to illustrate the applicability of our method beyond mere randomly built graphs.

We run our experiments on a large set of benchmarks, testing 135 different designs derived from Biere et al.. Figures 4.a and 4.b plot the error for all designs, and, for the sake of compactness, Tables 7 and 8

Table 6: Experimental results on “general” graphs with different characteristics.

n	m	Density [%]	Depth	Width	Chains	ϵ [%]	Time [s]	Memory [MB]
1002	10102	1.007	40	83	111	33.73	0.58	0.42
1002	20051	1.999	81	41	54	31.71	0.61	0.21
1002	100011	9.971	303	10	11	10.00	0.82	0.04
10002	18659	0.019	10	5244	5490	4.69	7.71	209.47
10002	24936	0.025	15	3477	3923	12.83	7.01	149.68
10002	52634	0.053	30	1601	2026	26.55	4.79	77.30
10002	100986	0.101	51	816	1085	32.97	5.34	41.40
10002	250138	0.250	115	334	445	33.23	4.34	16.98
10002	501785	0.502	225	170	223	31.18	2.47	8.51
50002	3123951	0.125	311	620	877	41.45	86.30	167.28
50002	6248250	0.250	580	320	447	39.69	69.39	85.26
75002	2342341	0.042	434	9254	9878	6.74	393.50	2826.19
100002	166694	0.002	9	41323	42257	2.26	1355.15	16120.09

report data for just 20 of them. Those 20 designs are selected as follows. The first 10 designs are the biggest ones in terms of number of gates (*e.g.*, graph vertices) our algorithm was able to manipulate in the allowed time limit. Those designs are reported in the first part of the two tables. The last 10 designs includes cases on which our algorithm presented the worst performance in terms of accuracy. Those designs are reported in the second part of the two tables.

More in details, Table 7 illustrate the characteristics of the benchmarks, *i.e.*, the number of primary inputs (#PIs), latches (#FFs), and gates (#ANDs). All benchmarks are single output circuits (being the output the property under verification), thus the number of primary outputs is omitted being always equal to 1. The number of primary inputs plus the number of latches determine the number of original sources of the derived graphs. Likewise, the number of primary outputs plus the number of latches determine the number of original sinks. Note that latches are modelled both as sources and sinks as they realize the sequential behaviour of the circuits. From a structural stand point, given the fact that the nodes within our derived graph can either originate from latches or logic gates, the in-degree of each node in the graph is at most two, whereas the out degree is unbounded^(iv).

Table 8 shows the results of our experiments. All columns have the same meaning previously introduced, but the first one, reporting the name of the benchmark. As it can be noticed, the largest instances

^(iv) Those benchmarks are usually stored in AIGER format, representing designs using AIGs (And-Inverter Graphs), *i.e.*, using as basic logic blocks only NOT and two-input AND gates.

Table 7: Details concerning AIG-derived circuits.

Benchmark	#PIs	#FFs	#ANDs
6s404rb1	202	9801	126011
6s349rb06	254	14090	118595
6s289rb05233	1085	12707	115953
6s119	579	18833	107093
6s290	543	13679	107242
6s330rb06	1056	7728	109695
6s344rb150	553	10669	87711
6s384rb024	7415	14952	65415
6s288r	12124	2461	71058
6s321b5	22	13126	66695
6s317b18	42	198	4849
6s357r	17	196	1234
6s268r	74	1324	7891
6s22	73	1126	17243
6s394r	17	180	930
6s335rb09	112	1658	10813
bobsmhdlc1	61	290	1626
6s399b02	47	98	9783
6s399b03	47	98	9783
beemrshr2f1	17	490	26068

have more than 125,000 vertices, and are managed in a few hundreds of seconds, obtaining extremely small errors.

Figures 4.a and 4.b visually represent the distribution of the number of chains with respect to the actual width of the graph and the relative error distribution, respectively. In Figure 4.a the grey area represents the optimum, whereas the overlying black area (almost invisible) represents the “surplus” due to the greedy approach. Figure 4.b plots the distribution of the error, sorted by increasing values, in percentage, in order to underline how many instances fall below a given error threshold.

The results presented shows that the number of chains found by our greedy approach is often very close to the optimum. The error percentage on the number of chains lies, in fact, within a 4% threshold for most of the instances. This suggests that our approach, despite being sub-optimal, can still be applied to large graph instances without incurring in excessive memory overhead, while significantly reducing computing time.

Table 8: Experimental results on graphs derived from circuits.

Graph	n	m	Density [%]	Depth	Width	Chains	ϵ [%]	Time [s]	Memory [MB]
6s404rb1	136016	271831	0.001	137	26937	27732	2.95	831.06	14389.02
6s289rb05233	129747	268956	0.002	28	42631	42773	0.33	1394.51	21170.30
6s119	126507	252431	0.002	111	37839	37915	0.20	927.02	18297.00
6s290	121466	242093	0.002	43	42429	42790	0.85	1104.13	19827.00
6s330rb06	118481	236015	0.002	50	37697	37889	0.51	919.47	17124.66
6s344rb150	98935	197712	0.002	53	39454	39700	0.62	802.52	14983.06
6s384rb024	87784	174298	0.002	31	38252	38419	0.44	601.57	12865.35
6s288r	85645	159161	0.002	108	24261	24313	0.21	365.26	7943.29
6s321b5	79845	159669	0.003	43	37231	37320	0.24	565.49	11367.09
6s317b18	5091	10167	0.039	45	1058	1140	7.75	1.37	22.14
6s357r	1449	2877	0.137	25	350	374	6.86	0.2	2.07
6s268r	9291	18495	0.021	35	2442	2609	6.84	4.58	92.47
6s22	18444	36811	0.011	132	3094	3305	6.82	9.69	232.53
6s394r	1129	2237	0.176	63	229	244	6.55	0.12	1.05
6s335rb09	12585	25081	0.016	43	4428	4675	5.58	8.04	244.44
bobsmdlc1	1979	3907	0.100	18	612	644	5.23	0.51	4.86
6s399b02	9930	19833	0.020	131	967	1012	4.65	2.29	38.33
beemrshr2f1	26577	52999	0.008	145	5832	6078	4.22	26.59	616.21

6 Conclusions

We proposed a novel greedy approach for computing the reachability index for DAGs. Our algorithm improves on existing methods with respect to time requirements, while relaxing the need of optimal memory consumption for storing such data structure. We provided an upper bound to the time and memory complexity of our approach. Our experimental analysis shows that the time-memory trade-off achieved by our algorithm is favourable, since our results are often very close to the optimum in terms of space requirements while significantly improving on computing time. Moreover, the computed errors tend to decrease with graphs of increasing size. We deem that our approach is profitable in contexts where reachability queries on large graphs must be answered but only a limited amount of time can be dedicated to pre-processing.

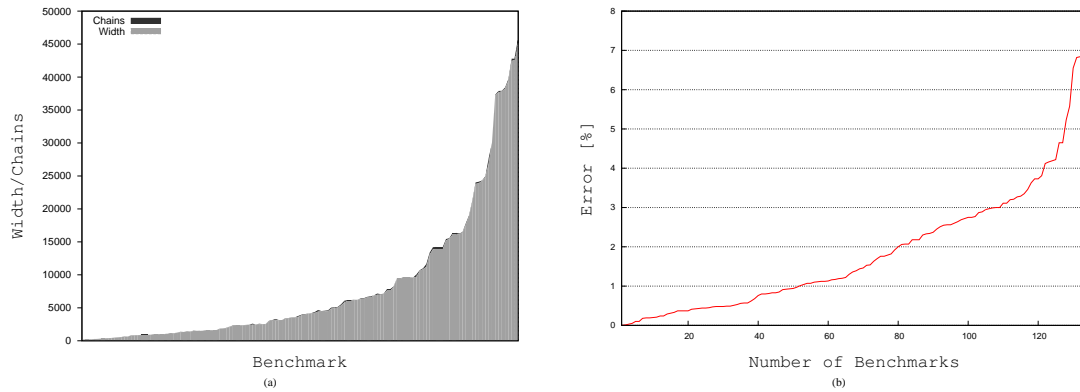


Figure 4: Visual representation on experiments using circuit-derived graphs: Number of chains compared to exact width of the graph (a), and distribution of the error w.r.t. to the optimum value (b).

References

- A. S. Asratian, T. M. J. Denley, and R. Häggkvist. *Bipartite Graphs and Their Applications*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-59345-X.
- J. Baumgartner, C. Loiacono, M. Palena, P. Pasini, D. Patti, S. Quer, S. Ricossa, and D. Vendramineto. Fast Cone-Of-Influence Computation and Estimation in Problems with Multiple Properties. In *Proc. Design Automation & Test in Europe Conf.*, pages 803–806, Grenoble, France, Mar. 2013. IEEE Computer Society.
- A. Biere, K. L. Claessen, and T. Jussila. The Hardware Model Checking Competition Web Page. Accessed: 2012-06-01.
- G. Cabodi, P. Camurati, and S. Quer. A graph-labeling approach for efficient cone-of-influence computation in model-checking problems with multiple properties. *Softw., Pract. Exper.*, 46(4):493–511, 2016. doi: 10.1002/spe.2321. URL <http://dx.doi.org/10.1002/spe.2321>.
- Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *IEEE 24th International Conference on Data Engineering, ICDE '08*, pages 893–902, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-1836-7. doi: 10.1109/ICDE.2008.4497498. URL <http://dx.doi.org/10.1109/ICDE.2008.4497498>.
- Y. Chen and Y. Chen. Decomposing DAGs into Spanning Trees: A new way to Compress Transitive Closures. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1007–1018, April 2011. doi: 10.1109/ICDE.2011.5767832.
- E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '02*, pages 937–946, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. ISBN 0-89871-513-X. URL <http://dl.acm.org/citation.cfm?id=545381.545503>.

- E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, USA, 1976.
- B. Edwards. NetworkX project Web Page. <https://networkx.github.io/>. Accessed: 2016-07-25.
- P. Erdős and A. Rényi. On random graphs i. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.
- S. Felsner, V. Raghavan, and J. Spinrad. Recognition algorithms for orders of small width and graphs of small dilworth number. *Order*, 20(4):351–364, 2003. ISSN 1572-9273. doi: 10.1023/B:ORDE.0000034609.99940.fb. URL <http://dx.doi.org/10.1023/B:ORDE.0000034609.99940.fb>.
- H. Wang and H. He and J. Yang and P. S. Yu and J. X. Yu. Dual Labeling: Answering Graph Reachability Queries in Constant Time. In *Proc. 22nd International Conference on Data Engineering (ICDE'06)*, page 75. IEEE Computer Society, 2006.
- H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, Dec. 1990. ISSN 0362-5915. doi: 10.1145/99935.99944. URL <http://doi.acm.org/10.1145/99935.99944>.
- R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently Answering Reachability Queries on Very Large Directed Graphs. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 595–608, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376677. URL <http://doi.acm.org/10.1145/1376616.1376677>.
- M. Kowaluk, A. Lingas, and J. Nowak. A path cover technique for lcas in dags. In J. Gudmundsson, editor, *Proc. of the 11th Scandinavian Workshop on Algorithm Theory, SWAT'2008 (Gothenburg, Sweden, July 2-4, 2008)*, volume 5124 of *LNCS*, pages 222–233. Springer-Verlag, Berlin-Heidelberg, 2008. URL http://dx.doi.org/10.1007/978-3-540-69903-3_21.
- J. Ruoming and W. Guan. Simple, fast, and scalable reachability oracle. *CoRR*, abs/1305.0502, 2013. URL <http://arxiv.org/abs/1305.0502>.
- P. Sanders and D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Algorithms, ESA 2005*, volume 3369 of *LNCS*. Springer, 2005.
- C. Shuchi. Course webpage. <http://pages.cs.wisc.edu/~shuchi/courses/880-S07/scribe-notes/lecture03.pdf>. Accessed: 2015-01-03.
- R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.
- J. Teuhola. Path signatures: A way to speed up recursion in relational databases. *IEEE Transaction on Knowledge and Data Engineering*, 8(3):446–454, June 1996. ISSN 1041-4347. doi: 10.1109/69.506711. URL <http://dx.doi.org/10.1109/69.506711>.

- S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 845–856, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-686-8. doi: 10.1145/1247480.1247573. URL <http://doi.acm.org/10.1145/1247480.1247573>.
- J. S. van Schaik and D. de Moor. A Memory Efficient Reachability Data Structure Through Bit Vector Compression. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 913–924, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0661-4. doi: 10.1145/1989323.1989419. URL <http://doi.acm.org/10.1145/1989323.1989419>.
- V. V. Vazirani. *Approximation Algorithms*. Springer, 2001.
- D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press.
- H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *Proc. VLDB Endow.*, 3(1-2):276–284, Sept. 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920879. URL <http://dx.doi.org/10.14778/1920841.1920879>.